UDACITY MACHINE LEARNING NANODEGREE 2019

CAPSTONE PROJECT

Classifying Urban sounds using Deep Learning

Abdelrhman Majdy

19/6/2019

1 Definition

1.1 Project Overview

Sounds are all around us. Whether directly or indirectly, we are always in contact with audio data. Sounds outline the context of our daily activities, ranging from the conversations we have
when interacting with people, the music we listen to, and all the other environmental sounds that we hear on a daily basis such as a car driving past, the patter of rain, or any other kind of background noise. The human brain is continuously processing and understanding this audio data, either consciously or subconsciously, giving us information about the environment around
us.
Automatic environmental sound classification is a growing area of research with numerous real
world applications. Whilst there is a large body of research in related audio fields such as speech
and music, work on the classification of environmental sounds is comparatively scarce. Likewise,
observing the recent advancements in the field of image classification where convolutional neural
networks are used to to classify images with high accuracy and at scale, it begs the question of the
applicability of these techniques in other domains, such as sound classification, where discrete
sounds happen over time.
The goal of this capstone project, is to apply Deep Learning techniques to the classification of environmental sounds, specifically focusing on the identification of particular urban sounds.
There is a plethora of real world applications for this research, such as:
• Content-based multimedia indexing and retrieval
• Assisting deaf individuals in their daily activities
• Smart home use cases such as 360-degree safety and security capabilities
• Automotive where recognising sounds both inside and outside of the car can improve safety
• Industrial uses such as predictive maintenance
My personal motivation for working on sound classification is my background in DSP and Audio
processing. Having worked on a number of projects in this field over the years, most recently at audio connectivity startup chirp.io, I am keen to apply my machine learning knowledge to this domain.

1.2 Problem Statement

The objective of this project will be to use Deep Learning techniques to classify urban sounds.
When given an audio sample in a computer readable format (such as a .wav file) of a few seconds
duration, we want to be able to determine if it contains one of the target urban sounds with a corresponding likelihood score. Conversely, if none of the target sounds were detected, we will be presented with an unknown score.
To achieve this, we plan on using different neural network architectures such as Multi-Layer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs).

1.3 Metrics

The evaluation metric for this problem will be the 'Classification Accuracy' which is defined asthe percentage of correct predictions.
Accuracy = correct classifications / number of classifications
Classification Accuracy was deemed to be the optimal choice metric as it is presumed that the dataset will be relatively symmetrical (as we will explore in the next section) with this being a

multi-class classifier whereby the target data classes will be generally uniform in size.
Other metrics such as Precision, Recall (or combined as the F1 score) were ruled out as they are more applicable to classification challenges that contain a relatively tiny target class in an unbalanced data set.

2 Analysis

2.1 Data Exploration and Visualisation

2.1.1 UrbanSound dataset

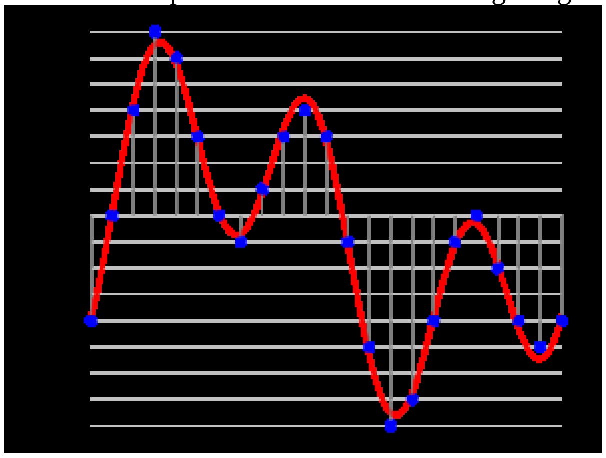
For this project we will use a dataset called Urbansound8K. The dataset contains 8732 sound excerpts
(<=4s) of urban sounds from 10 classes, which are:

- Air Conditioner
- Car Horn
- Children Playing
- Dog bark
- Drilling
- Engine Idling
- Gun Shot
- Jackhammer
- Siren
- Street Music

The accompanying metadata contains a unique ID for each sound excerpt along with it’s given class name.
A sample of this dataset is included with the accompanying git repo and the full dataset can be downloaded from here.

2.1.2 Audio sample file data overview

These sound excerpts are digital audio files in .wav format.
Sound waves are digitised by sampling them at discrete intervals known as the sampling rate (typically 44.1kHz for CD quality audio meaning samples are taken 44,100 times per second).
Each sample is the amplitude of the wave at a particular time interval, where the bit depth determines how detailed the sample will be also known as the dynamic range of the signal (typically 16bit which means a sample can range from 65,536 amplitude values).
This can be represented with the following image:



Therefore, the data we will be analysing for each sound excerpts is essentially a one dimensional array or vector of amplitude values.

2.1.3 Analysing audio data

For audio analysis, we will be using the following libraries:
1. IPython.display.Audio This allows us to play audio directly in the Jupyter Notebook.
2. Librosa librosa is a Python package for music and audio processing by Brian McFee and will allow us to load audio in our notebook as a numpy array for analysis and manipulation.
You may need to install librosa using pip as follows:
pip install librosa

2.1.4 Auditory inspection

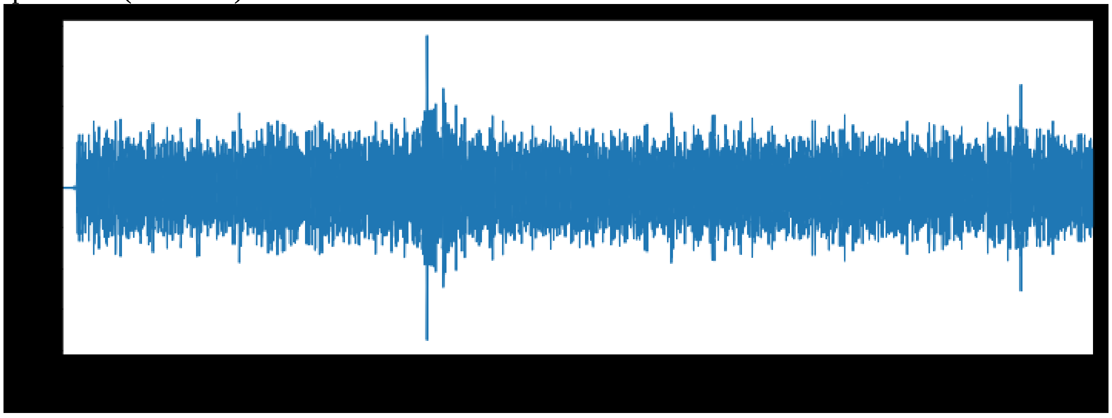
We will use IPython.display.Audio to play the audio files so we can inspect aurally.

```
import IPython.display as ipd
ipd.Audio('../UrbanSound Dataset sample/audio/100032-3-0-0.wav')
```

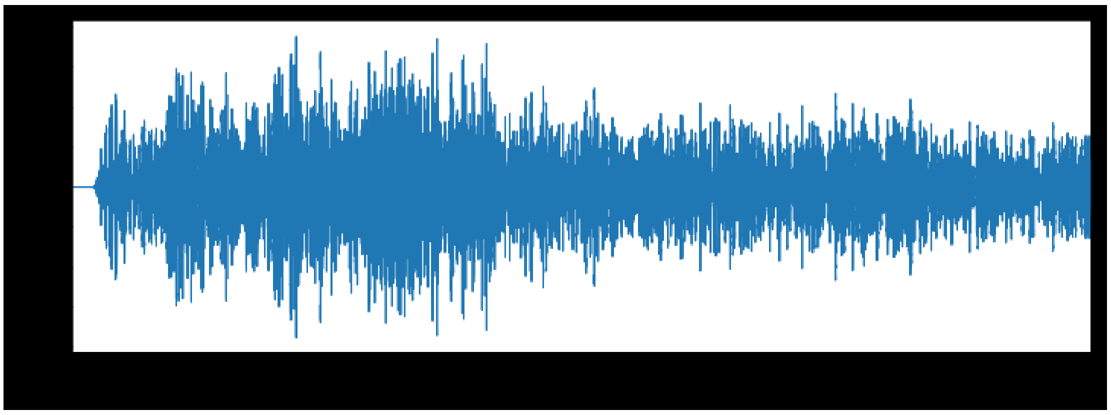
2.1.5 Visual inspection

We will load a sample from each class and visually inspect the data for any patterns. We will use librosa to load the audio file into an array then librosa.display and matplotlib to display the waveform.

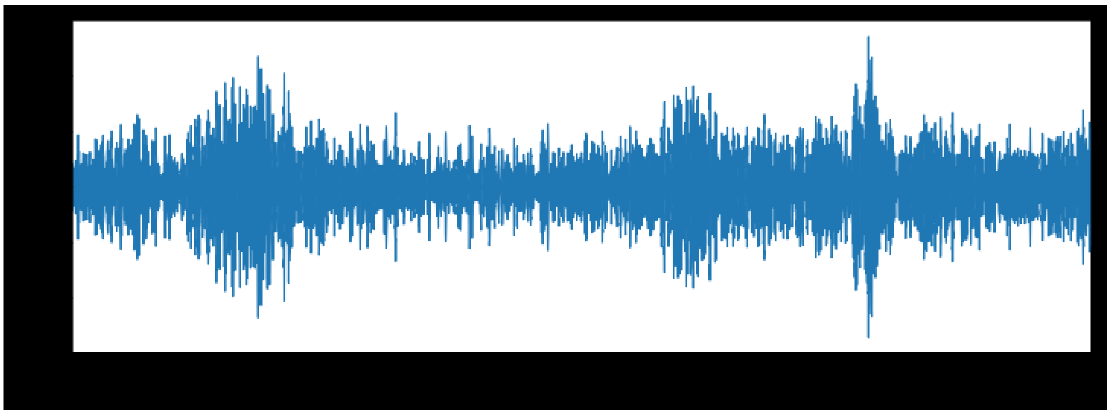
```
# Load imports
import IPython.display as ipd
import librosa
import librosa.display
import matplotlib.pyplot as plt
# Class: Air Conditioner
filename = '../UrbanSound Dataset sample/audio/100852-0-0-0.wav'
plt.figure(figsize=(12,4))
data,sample_rate = librosa.load(filename)
_ = librosa.display.waveplot(data,sr=sample_rate)
ipd.Audio(filename)
```

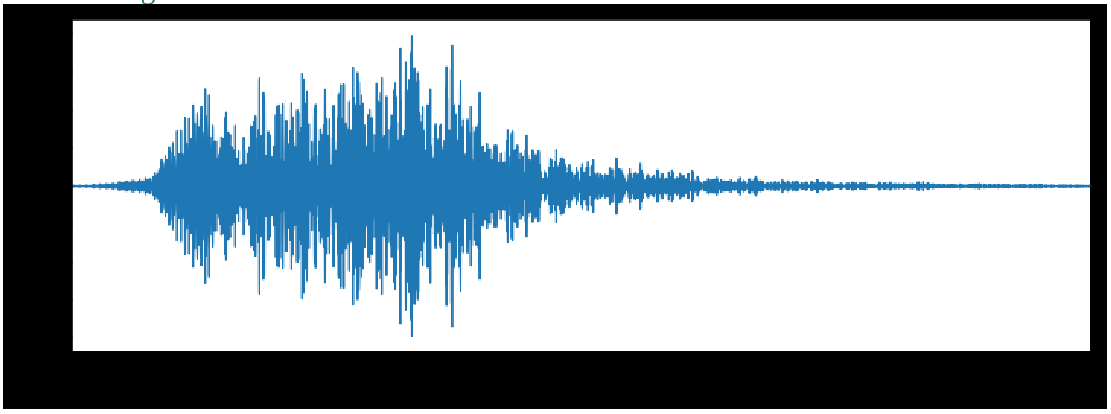


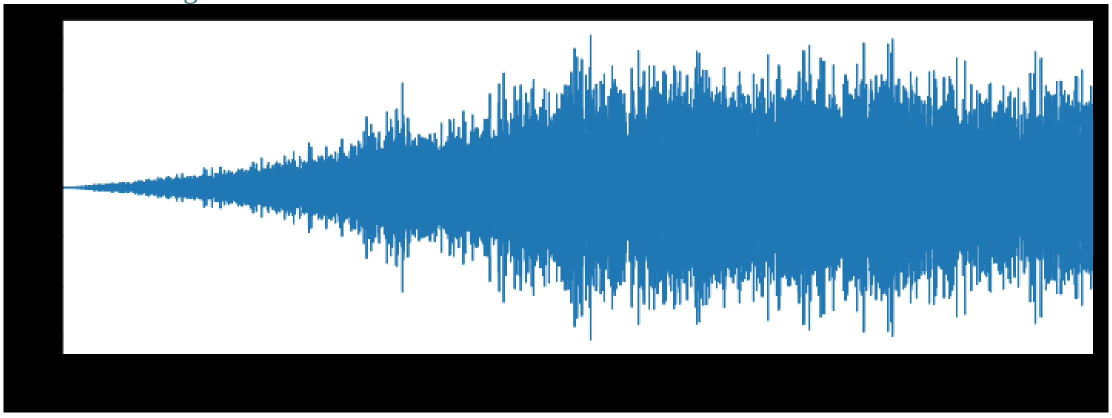
```
# Class: Car horn
```

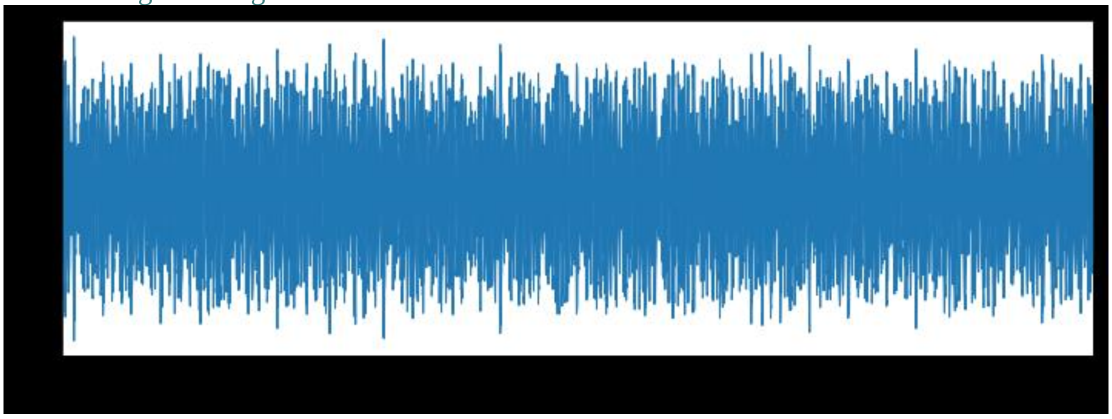


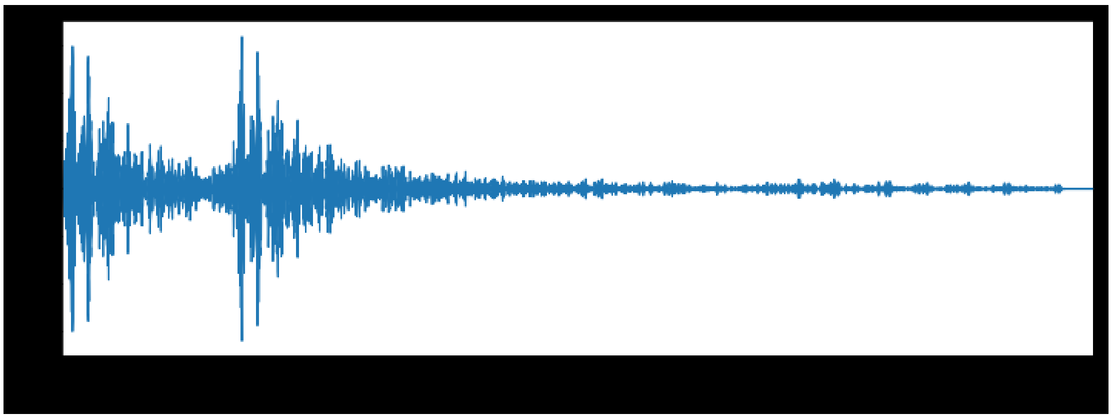
```
# Class: Children playing
```


# Class: Dog bark

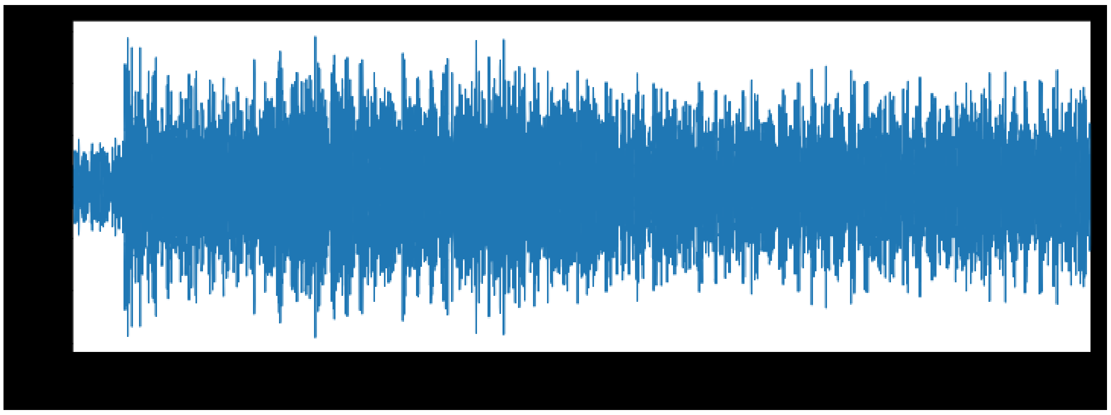

# Class: Drilling


# Class: Engine Idling

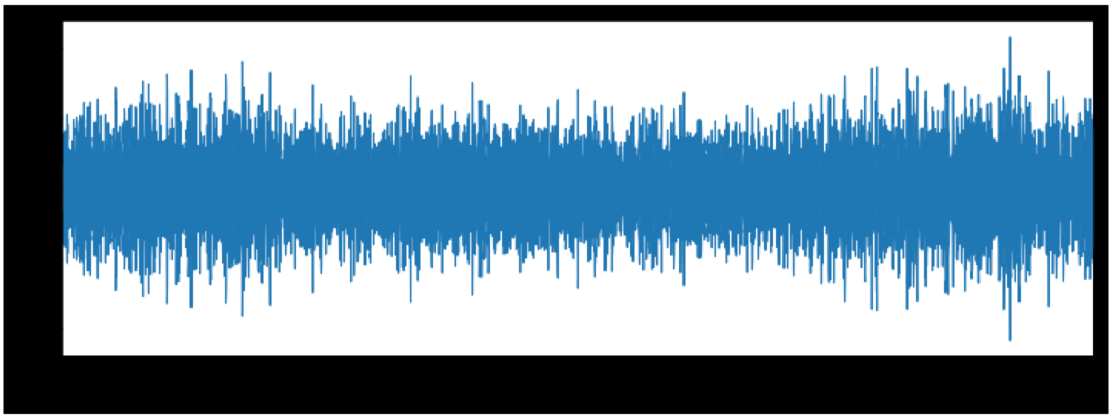

# Class: Gunshot



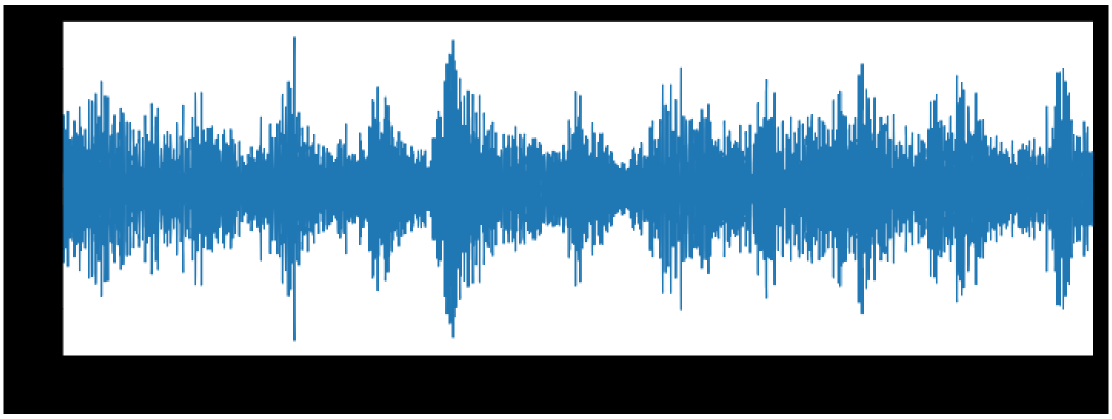
# Class: Jackhammer



# Class: Siren



# Class: Street music



2.1.6 Observations
From a visual inspection we can see that it is tricky to visualise the difference between some of the classes.
Particularly, the waveforms for repetitive sounds for air conditioner, drilling, engine idling and jackhammer are similar in shape.
Likewise the peak in the dog barking sample is similar in shape to the gun shot sample (albeit the samples differ in that there are two peaks for two gunshots compared to the one peak for one dog bark). Also, the car horn is similar too. There are also similarities between the children playing and street music.
The human ear can naturally detect the difference between the harmonics, it will be interesting to see how well a deep learning model will be able to extract the necessary features to distinguish between these classes.
However, it is easy to differentiate from the waveform shape, the difference between certain classes such as dog barking and jackhammer.
2.1.7 Dataset Metadata
Here we will load the UrbanSound metadata .csv file into a Panda dataframe.
```
import pandas as pd
metadata = pd.read_csv('../UrbanSound Dataset sample/metadata/UrbanSound8K.csv')
metadata.head()
```
```
slice_file_name fsID start end salience fold classID \ class_name
0 100032-3-0-0.wav 100032 0.0 0.317551 1 5 3 dog_bark
1 100263-2-0-117.wav 100263 58.5 62.500000 1 5 2 children_playing
2 100263-2-0-121.wav 100263 60.5 64.500000 1 5 2 children_playing
3 100263-2-0-126.wav 100263 63.0 67.000000 1 5 2 children_playing
4 100263-2-0-137.wav 100263 68.5 72.500000 1 5 2 children_playing
```
2.1.8 Class distributions
```
print(metadata.class_name.value_counts())
```
```
children_playing 1000
dog_bark 1000
street_music 1000
jackhammer 1000
engine_idling 1000
air_conditioner 1000
drilling 1000
siren 929
car_horn 429
gun_shot 374
```
2.1.9 Observations
Here we can see the Class labels are unbalanced. Although 7 out of the 10 classes all have exactly 1000 samples, and siren is not far off with 929, the remaining two (car_horn, gun_shot) havesignificantly less samples at 43% and 37% respectively.
This will be a concern and something we may need to address later on.
2.1.10 Audio sample file properties
Next we will iterate through each of the audio sample files and extract, number of audio channels,
sample rate and bit-depth.
```
# Load various imports
import pandas as pd
import os
import librosa
import librosa.display
from helpers.wavfilehelper import WavFileHelper
wavfilehelper = WavFileHelper()
audiodata = []
for index, row in metadata.iterrows():
```

```
file_name = os.path.join(os.path.abspath('/Volumes/Untitled/ML_Data/Urban
Sound/UrbanSound8K/data = wavfilehelper.read_file_properties(file_name)
audiodata.append(data)
# Convert into a Panda dataframe
audiodf = pd.DataFrame(audiodata, columns=['num_channels','sample_rate','bit_depth'])
```

2.1.11 Audio channels

Most of the samples have two audio channels (meaning stereo) with a few with just the one channel (mono).
The easiest option here to make them uniform will be to merge the two channels in the stereo samples into one by averaging the values of the two channels.

```
# num of channels
print(audiodf.num_channels.value_counts(normalize=True))
2 0.915369
1 0.084631
```

2.1.12 Sample rate

There is a wide range of Sample rates that have been used across all the samples which is a concern (ranging from 96k to 8k).
This likely means that we will have to apply a sample-rate conversion technique (either upconversion
or down-conversion) so we can see an agnostic representation of their waveform which will allow us to do a fair comparison.

```
# sample rates
print(audiodf.sample_rate.value_counts(normalize=True))
44100 0.614979
48000 0.286532
96000 0.069858
24000 0.009391
16000 0.005153
22050 0.005039
11025 0.004466
192000 0.001947
8000 0.001374
11024 0.000802
32000 0.000458
```

2.1.13 Bit-depth

There is also a wide range of bit-depths. It's likely that we may need to normalise them by taking
the maximum and minimum amplitude values for a given bit-depth.

```
# bit depth
print(audiodf.bit_depth.value_counts(normalize=True))
16 0.659414
24 0.315277
32 0.019354
8 0.004924
4 0.001031
```

2.1.14 Other audio properties to consider

We may also need to consider normalising the volume levels (wave amplitude value) if this is seen
to vary greatly, by either looking at the peak volume or the RMS volume.

2.2 Algorithms and Techniques

The proposed solution to this problem is to apply Deep Learning techniques that have proved to
be highly successful in the field of image classification.
First we will extract Mel-Frequency Cepstral Coefficients (MFCC) [2] from the the audio samples

on a per-frame basis with a window size of a few milliseconds. The MFCC summarises the frequency distribution across the window size, so it is possible to analyse both the frequency and
time characteristics of the sound. These audio representations will allow us to identify features
for classification.
The next step will be to train a Deep Neural Network with these data sets and make predictions.
We will begin by using a simple neural network architecture, such as Multi-Layer Perceptron before experimenting with more complex architectures such as Convolutional Neural Networks.
Multi-layer perceptron's (MLP) are classed as a type of Deep Neural Network as they are composed of more than one layer of perceptrons and use non-linear activation which distinguish
them from linear perceptrons. Their architecture consists of an input layer, an output layer that
ultimately make a prediction about the input, and in-between the two layers there is an arbitrary
number of hidden layers.
These hidden layers have no direct connection with the outside world and perform the model computations. The network is fed a labelled dataset (this being a form of supervised learning) of
input-output pairs and is then trained to learn a correlation between those inputs and outputs.
The training process involves adjusting the weights and biases within the perceptrons in the hidden layers in order to minimise the error.
The algorithm for training an MLP is known as Backpropagation. Starting with all weights in the network being randomly assigned, the inputs do a forward pass through the network and the decision of the output layer is measured against the ground truth of the labels you want to predict. Then the weights and biases are backpropagated back though the network where an optimisation method, typically Stochastic Gradient descent is used to adjust the weights so they
will move one step closer to the error minimum on the next pass. The training phase will keep on performing this cycle on the network until it the error can go no lower which is known as convergence.
Convolutional Neural Networks (CNNs) build upon the architecture of MLPs but with a number
of important changes. Firstly, the layers are organised into three dimensions, width, height and
depth. Secondly, the nodes in one layer do not necessarily connect to all nodes in the subsequent
layer, but often just a sub region of it.
This allows the CNN to perform two important stages. The first being the feature extraction phase. Here a filter window slides over the input and extracts a sum of the convolution at each location which is then stored in the feature map. A pooling process is often included between CNN layers where typically the max value in each window is taken which decreases the feature
map size but retains the significant data. This is important as it reduces the dimensionality of the
network meaning it reduces both the training time and likelihood of overfitting. Then lastly we
have the classification phase. This is where the 3D data within the network is flattened into a 1D vector to be output. For the reasons discussed, both MLPs and CNN's typically make good classifiers, where CNN's in particular perform very well with image classification tasks due to their feature extraction and classification parts. I believe that this will be very effective

at finding patterns within the MFCC's much like they are effective at finding patterns within images.
We will use the evaluation metrics described in earlier sections to compare the performance of these solutions against the benchmark models in the next section.

2.3 Benchmark Model

For the benchmark model, we will use the algorithms outlined in the paper "A Dataset and Taxonomy
for Urban Sound Research" (Salamon, 2014) [3]. The paper describes five different algorithms
with the following accuracies for a audio slice maximum duration of 4 seconds using the same UrbanSound
dataset.

Algorithm	**Classification Accuracy**
SVM_rbf	68%
RandomForest500	66%
IBk5	55%
J48	48%
ZeroR	10%

3 Methodology

3.1 Data Preprocessing and Data Splitting

3.1.1 Audio properties that will require normalising

Following on from the previous section, we identified the following audio properties that need
preprocessing to ensure consistency across the whole dataset:
• Audio Channels
• Sample rate
• Bit-depth
We will continue to use Librosa which will be useful for the pre-processing and feature extraction.

3.1.2 Preprocessing stage

For much of the preprocessing we will be able to use Librosa's load() function.
We will compare the outputs from Librosa against the default outputs of scipy's wavfile library
using a chosen file from the dataset.
Sample rate conversion By default, Librosa's load function converts the sampling rate to 22.05
KHz which we can use as our comparison level.

```
import librosa
from scipy.io import wavfile as wav
import numpy as np
filename = '../UrbanSound Dataset sample/audio/100852-0-0-0.wav'
librosa_audio, librosa_sample_rate = librosa.load(filename)
scipy_sample_rate, scipy_audio = wav.read(filename)
print('Original sample rate:', scipy_sample_rate)
print('Librosa sample rate:', librosa_sample_rate)
```

```
Original sample rate: 44100
Librosa sample rate: 22050
```

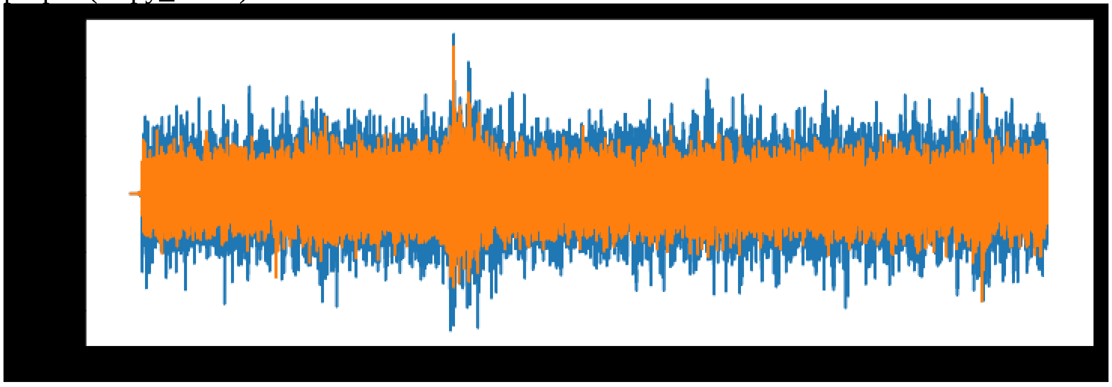
Bit-depth Librosa's load function will also normalise the data so it's values range between -1 and 1. This removes the complication of the dataset having a wide range of bit-depths.

```
print('Original audio file min~max range:', np.min(scipy_audio), 'to', np.max(scipy_audio))
print('Librosa audio file min~max range:', np.min(librosa_audio), 'to', np.max(librosa_audio))
```

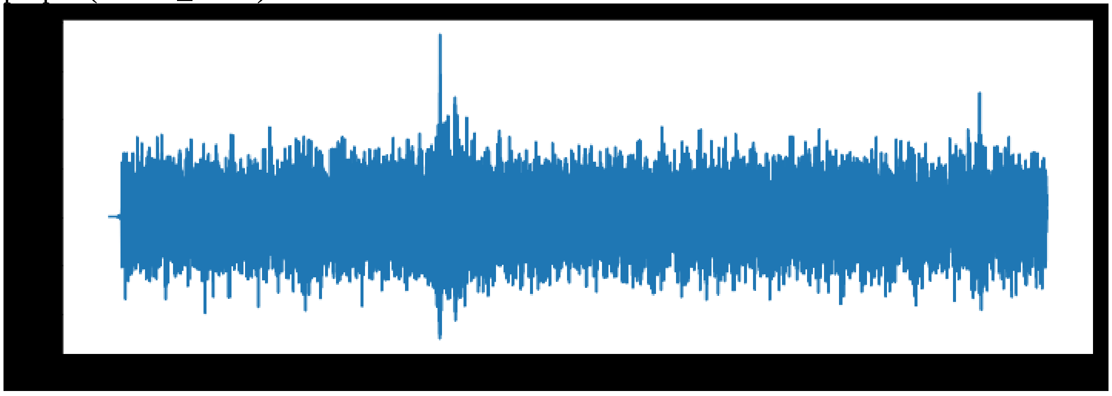
```
Original audio file min~max range: -23628 to 27507
Librosa audio file min~max range: -0.50266445 to 0.74983937
```

Merge audio channels Librosa will also convert the signal to mono, meaning the number of channels will always be 1.

```
import matplotlib.pyplot as plt
# Original audio with 2 channels
plt.figure(figsize=(12, 4))
plt.plot(scipy_audio)
```



```
# Librosa audio with channels merged
plt.figure(figsize=(12, 4))
plt.plot(librosa_audio)
```



Other audio properties to consider At this stage it is not yet clear whether other factors may also need to be taken into account, such as sample duration length and volume levels.
We will proceed as is for the meantime and come back to address these later if it's perceived to be
effecting the validity of our target metrics.

3.1.3 Extract Features

As outlined in the proposal, we will extract Mel-Frequency Cepstral Coefficients (MFCC) from
the the audio samples.
The MFCC summarises the frequency distribution across the window size, so it is possible to analyse both the frequency and time characteristics of the sound. These audio representations will allow us to identify features for classification.
Extracting a MFCC For this we will use Librosa's mfcc() function which generates an MFCC
from time series audio data.

```
mfccs = librosa.feature.mfcc(y=librosa_audio, sr=librosa_sample_rate, n_mfcc=40)
print(mfccs.shape)
(40, 173)
```

This shows librosa calculated a series of 40 MFCCs over 173 frames.

```
import librosa.display
librosa.display.specshow(mfccs, sr=librosa_sample_rate, x_axis='time')
```

Extracting MFCC's for every file We will now extract an MFCC for each audio file in the dataset
and store it in a Panda Dataframe along with it's classification label.

```
def extract_features(file_name):
try:
audio, sample_rate = librosa.load(file_name, res_type='kaiser_fast')
mfccs = librosa.feature.mfcc(y=audio, sr=sample_rate, n_mfcc=40)
mfccsscaled = np.mean(mfccs.T,axis=0)
except Exception as e:
print("Error encountered while parsing file: ", file)
return None
return mfccsscaled
# Load various imports
import pandas as pd
import os
import librosa
# Set the path to the full UrbanSound dataset
fulldatasetpath = '/Volumes/Untitled/ML_Data/Urban Sound/UrbanSound8K/audio/'
metadata = pd.read_csv('../UrbanSound Dataset sample/metadata/UrbanSound8K.csv')
features = []
# Iterate through each sound file and extract the features
for index, row in metadata.iterrows():
file_name =
os.path.join(os.path.abspath(fulldatasetpath),'fold'+str(row["fold"])+'/',str(row["class_label =
row["class_name"]
data = extract_features(file_name)
features.append([data, class_label])
# Convert into a Panda dataframe
featuresdf = pd.DataFrame(features, columns=['feature','class_label'])
print('Finished feature extraction from ', len(featuresdf), ' files')
```

```
Finished feature extraction from 8732 files
```

3.1.4 Convert the data and labels

We will use sklearn.preprocessing.LabelEncoder to encode the categorical text data into model-understandable numerical data.

```
from sklearn.preprocessing import LabelEncoder
from keras.utils import to_categorical
# Convert features and corresponding classification labels into numpy arrays
X = np.array(featuresdf.feature.tolist())
y = np.array(featuresdf.class_label.tolist())
```

```
# Encode the classification labels
le = LabelEncoder()
yy = to_categorical(le.fit_transform(y))
```

3.1.5 Split the dataset

Here we will use sklearn.model_selection.train_test_split to split the dataset into training and testing sets. The testing set size will be 20% and we will set a random state.

```
# split the dataset
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X, yy, test_size=0.2, random_state = 42)
```

3.2 Implementation

3.2.1 Initial model architecture - MLP

We will start with constructing a Multilayer Perceptron (MLP) Neural Network using Keras and
a Tensorflow backend.
Starting with a sequential model so we can build the model layer by layer.
We will begin with a simple model architecture, consisting of three layers, an input layer, a hidden
layer and an output layer. All three layers will be of the dense layer type which is a standard layer type that is used in many cases for neural networks.
The first layer will receive the input shape. As each sample contains 40 MFCCs (or columns) we
have a shape of (1x40) this means we will start with an input shape of 40.
The first two layers will have 256 nodes. The activation function we will be using for our first 2
layers is the ReLU, or Rectified Linear Activation. This activation function has been proven to
work well in neural networks.
We will also apply a Dropout value of 50% on our first two layers. This will randomly exclude
nodes from each update cycle which in turn results in a network that is capable of better generalisation
and is less likely to overfit the training data.
Our output layer will have 10 nodes (num_labels) which matches the number of possible classifications.
The activation is for our output layer is softmax. Softmax makes the output sum up to 1
so the output can be interpreted as probabilities. The model will then make its prediction based
on which option has the highest probability.

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.optimizers import Adam
from keras.utils import np_utils
from sklearn import metrics
num_labels = yy.shape[1]
filter_size = 2
# Construct model
model = Sequential()
model.add(Dense(256, input_shape=(40,)))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(num_labels))
model.add(Activation('softmax'))
```

3.2.2 Compiling the model

For compiling our model, we will use the following three parameters:

• Loss function - we will use categorical_crossentropy. This is the most common choice for classification. A lower score indicates that the model is performing better.

• Metrics - we will use the accuracy metric which will allow us to view the accuracy score on the validation data when we train the model.

• Optimizer - here we will use adam which is a generally good optimizer for many use cases.

```
# Compile the model
model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='adam')
# Display model architecture summary
model.summary()
# Calculate pre-training accuracy
score = model.evaluate(x_test, y_test, verbose=0)
accuracy = 100*score[1]
print("Pre-training accuracy: %.4f%%" % accuracy)
```

```
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
dense_1 (Dense) (None, 256) 10496
_________________________________________________________________
activation_1 (Activation) (None, 256) 0
_________________________________________________________________
dropout_1 (Dropout) (None, 256) 0
_________________________________________________________________
dense_2 (Dense) (None, 256) 65792
_________________________________________________________________
activation_2 (Activation) (None, 256) 0
_________________________________________________________________
dropout_2 (Dropout) (None, 256) 0
_________________________________________________________________
dense_3 (Dense) (None, 10) 2570
_________________________________________________________________
activation_3 (Activation) (None, 10) 0
=================================================================
Total params: 78,858
Trainable params: 78,858
Non-trainable params: 0
_________________________________________________________________
Pre-training accuracy: 11.5627%
```

3.2.3 Training

Here we will train the model.

We will start with 100 epochs which is the number of times the model will cycle through the data.

The model will improve on each cycle until it reaches a certain point.

We will also start with a low batch size, as having a large batch size can reduce the generalisation
ability of the model.

```
from keras.callbacks import ModelCheckpoint
from datetime import datetime
num_epochs = 100
num_batch_size = 32
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.basic_mlp.hdf5',
verbose=1, save_best_only=True)
```

```
start = datetime.now()
model.fit(x_train, y_train, batch_size=num_batch_size, epochs=num_epochs,
validation_data=(x_test, duration = datetime.now() - start
print("Training completed in time: ", duration)
Train on 6985 samples, validate on 1747 samples
Epoch 00097: val_loss did not improve from 0.42049
Epoch 98/100
6985/6985 [==============================] - 2s 329us/step - loss: 0.5246 - acc:
0.8241 - val_loss: Epoch 00098: val_loss did not improve from 0.42049
Epoch 99/100
6985/6985 [==============================] - 2s 347us/step - loss: 0.5346 - acc:
0.8169 - val_loss: Epoch 00099: val_loss did not improve from 0.42049
Epoch 100/100
6985/6985 [==============================] - 2s 351us/step - loss: 0.5413 - acc:
0.8153 - val_loss: Epoch 00100: val_loss did not improve from 0.42049
Training completed in time: 0:04:15.582298
```

3.2.4 Test the model

Here we will review the accuracy of the model on both the training and test data sets.

```
# Evaluating the model on the training and testing set
score = model.evaluate(x_train, y_train, verbose=0)
print("Training Accuracy: ", score[1])
score = model.evaluate(x_test, y_test, verbose=0)
print("Testing Accuracy: ", score[1])
Training Accuracy: 0.9252684323550465
Testing Accuracy: 0.8763594734511787
```

The initial Training and Testing accuracy scores are quite high. As there is not a great difference
between the Training and Test scores (~5%) this suggests that the model has not suffered from
overfitting.

3.2.5 Predictions

Here we will build a method which will allow us to test the models predictions on a specified audio .wav file.

```
import librosa
import numpy as np
def extract_feature(file_name):
try:
audio_data, sample_rate = librosa.load(file_name, res_type='kaiser_fast')
mfccs = librosa.feature.mfcc(y=audio_data, sr=sample_rate, n_mfcc=40)
mfccsscaled = np.mean(mfccs.T,axis=0)
except Exception as e:
print("Error encountered while parsing file: ", file)
return None, None
return np.array([mfccsscaled])
def print_prediction(file_name):
prediction_feature = extract_feature(file_name)
predicted_vector = model.predict_classes(prediction_feature)
predicted_class = le.inverse_transform(predicted_vector)
print("The predicted class is:", predicted_class[0], '\n')
predicted_proba_vector = model.predict_proba(prediction_feature)
predicted_proba = predicted_proba_vector[0]
for i in range(len(predicted_proba)):
category = le.inverse_transform(np.array([i]))
print(category[0], "\t\t : ", format(predicted_proba[i], '.32f') )
```

3.2.6 Validation

Test with sample data Initial sanity check to verify the predictions using a subsection of the sample audio files we explored in the first notebook. We expect the bulk of these to be classified
correctly.

```
# Class: Air Conditioner
filename = '../UrbanSound Dataset sample/audio/100852-0-0-0.wav'
print_prediction(filename)
The predicted class is: air_conditioner
# Class: Drilling
filename = '../UrbanSound Dataset sample/audio/103199-4-0-0.wav'
print_prediction(filename)
The predicted class is: drilling
# Class: Street music
filename = '../UrbanSound Dataset sample/audio/101848-9-0-0.wav'
print_prediction(filename)
The predicted class is: street_music
# Class: Car Horn
filename = '../UrbanSound Dataset sample/audio/100648-1-0-0.wav'
print_prediction(filename)
The predicted class is: car_horn
```

Observations From this brief sanity check the model seems to predict well. One error was observed whereby a car horn was incorrectly classified as a dog bark.
We can see from the per class confidence that this was quite a low score (43%). This allows follows our early observation that a dog bark and car horn are similar in spectral shape.

3.2.7 Other audio

Here we will use a sample of various copyright free sounds that we not part of either our test or training data to further validate our model.

```
filename = '../Evaluation audio/dog_bark_1.wav'
print_prediction(filename)
The predicted class is: dog_bark
filename = '../Evaluation audio/drilling_1.wav'
print_prediction(filename)
The predicted class is: drilling
filename = '../Evaluation audio/gun_shot_1.wav'
print_prediction(filename)
# sample data weighted towards gun shot - peak in the dog barking sample is simmilar in
# shape to the gun shot sample.
The predicted class is: dog_bark
filename = '../Evaluation audio/siren_1.wav'
print_prediction(filename)
The predicted class is: siren
```

Observations The performance of our initial model is satisfactory and has generalised well, seeming to predict well when tested against new audio data.

3.3 Refinement

In our initial attempt, we were able to achieve a Classification Accuracy score of:

• Training data Accuracy: 92.3%
• Testing data Accuracy: 87%

We will now see if we can improve upon that score using a Convolutional Neural Network (CNN).

Feature Extraction refinement In the previous feature extraction stage, the MFCC vectors would vary in size for the different audio files (depending on the samples duration). However, CNNs require a fixed size for all inputs which means we will have to revisit the feature extraction code that we previously wrote. To overcome this we will zero pad the output vectors to make them all the same size.

```
import numpy as np
max_pad_len = 174
def extract_features(file_name):
try:
audio, sample_rate = librosa.load(file_name, res_type='kaiser_fast')
mfccs = librosa.feature.mfcc(y=audio, sr=sample_rate, n_mfcc=40)
pad_width = max_pad_len - mfccs.shape[1]
mfccs = np.pad(mfccs, pad_width=((0, 0), (0, pad_width)), mode='constant')
except Exception as e:
print("Error encountered while parsing file: ", file_name)
return None
return mfccs
# Load various imports
import pandas as pd
import os
import librosa
# Set the path to the full UrbanSound dataset
fulldatasetpath = '/Volumes/Untitled/ML_Data/Urban Sound/UrbanSound8K/audio/'
metadata = pd.read_csv('../UrbanSound Dataset sample/metadata/UrbanSound8K.csv')
features = []
# Iterate through each sound file and extract the features
for index, row in metadata.iterrows():
file_name =
os.path.join(os.path.abspath(fulldatasetpath),'fold'+str(row["fold"])+'/',str(row["class_label =
row["class_name"]
data = extract_features(file_name)
features.append([data, class_label])
# Convert into a Panda dataframe
featuresdf = pd.DataFrame(features, columns=['feature','class_label'])
print('Finished feature extraction from ', len(featuresdf), ' files')
Finished feature extraction from 8732 files
from sklearn.preprocessing import LabelEncoder

from keras.utils import to_categorical
# Convert features and corresponding classification labels into numpy arrays
X = np.array(featuresdf.feature.tolist())
y = np.array(featuresdf.class_label.tolist())
# Encode the classification labels
le = LabelEncoder()
yy = to_categorical(le.fit_transform(y))
# split the dataset
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X, yy, test_size=0.2, random_state = 42)
```

3.3.1 Convolutional Neural Network (CNN) model architecture

We will modify our model to be a Convolutional Neural Network (CNN) again using Keras and a Tensorflow backend.

Again we will use a sequential model, starting with a simple model architecture, consisting of four Conv2D convolution layers, with our final output layer being a dense layer.

The convolution layers are designed for feature detection. It works by sliding a filter window over the input and performing a matrix multiplication and storing the result in a feature map. This operation is known as a convolution.

The filter parameter specifies the number of nodes in each layer. Each layer will increase in size from 16, 32, 64 to 128, while the kernel_size parameter specifies the size of the kernel window which in this case is 2 resulting in a 2x2 filter matrix.

The first layer will receive the input shape of (40, 174, 1) where 40 is the number of MFCC's 174 is the number of frames taking padding into account and the 1 signifying that the audio is mono.
The activation function we will be using for our convolutional layers is ReLU which is the same as
our previous model. We will use a smaller Dropout value of 20% on our convolutional layers. Each convolutional layer has an associated pooling layer of MaxPooling2D type with the final convolutional layer having a GlobalAveragePooling2D type. The pooling layer is do reduce the
dimensionality of the model (by reducing the parameters and subsquent computation requirements)
which serves to shorten the training time and reduce overfitting. The Max Pooling type takes the maximum size for each window and the Global Average Pooling type takes the average
which is suitable for feeding into our dense output layer.
Our output layer will have 10 nodes (num_labels) which matches the number of possible classifications.
The activation is for our output layer is softmax. Softmax makes the output sum up to 1 so the output can be interpreted as probabilities. The model will then make its prediction based on which option has the highest probability.

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, Conv2D, MaxPooling2D,
GlobalAveragePooling2D
from keras.optimizers import Adam
from keras.utils import np_utils
from sklearn import metrics
num_rows = 40
num_columns = 174
num_channels = 1
x_train = x_train.reshape(x_train.shape[0], num_rows, num_columns, num_channels)
x_test = x_test.reshape(x_test.shape[0], num_rows, num_columns, num_channels)
num_labels = yy.shape[1]
filter_size = 2
# Construct model
model = Sequential()
model.add(Conv2D(filters=16, kernel_size=2, input_shape=(num_rows, num_columns,
num_channels), activation='model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.2))
model.add(Conv2D(filters=32, kernel_size=2, activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.2))
model.add(Conv2D(filters=64, kernel_size=2, activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.2))
model.add(Conv2D(filters=128, kernel_size=2, activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.2))
model.add(GlobalAveragePooling2D())
model.add(Dense(num_labels, activation='softmax'))
```

3.3.2 Compiling the model

For compiling our model, we will use the same three parameters as the previous model:

```
# Compile the model
```

```
model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='adam')
# Display model architecture summary
model.summary()

# Calculate pre-training accuracy
score = model.evaluate(x_test, y_test, verbose=1)
accuracy = 100*score[1]
print("Pre-training accuracy: %.4f%%" % accuracy)
```

```
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
conv2d_11 (Conv2D) (None, 39, 173, 16) 80
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 19, 86, 16) 0
_________________________________________________________________
dropout_17 (Dropout) (None, 19, 86, 16) 0
_________________________________________________________________
conv2d_12 (Conv2D) (None, 18, 85, 32) 2080
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 9, 42, 32) 0
_________________________________________________________________
dropout_18 (Dropout) (None, 9, 42, 32) 0
_________________________________________________________________
conv2d_13 (Conv2D) (None, 8, 41, 64) 8256
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 4, 20, 64) 0
_________________________________________________________________
dropout_19 (Dropout) (None, 4, 20, 64) 0
_________________________________________________________________
conv2d_14 (Conv2D) (None, 3, 19, 128) 32896
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 1, 9, 128) 0
_________________________________________________________________
dropout_20 (Dropout) (None, 1, 9, 128) 0
_________________________________________________________________
global_average_pooling2d_1 ( (None, 128) 0
_________________________________________________________________
dense_13 (Dense) (None, 10) 1290
=================================================================
Total params: 44,602
Trainable params: 44,602
Non-trainable params: 0
_________________________________________________________________
1747/1747 [==============================] - 9s 5ms/step
Pre-training accuracy: 12.0206%
```

3.3.3 Training

Here we will train the model. As training a CNN can take a sigificant amount of time, we will start with a low number of epochs and a low batch size. If we can see from the output that the model is converging, we will increase both numbers.

```
from keras.callbacks import ModelCheckpoint
from datetime import datetime
#num_epochs = 12
#num_batch_size = 128
num_epochs = 72
```

```
num_batch_size = 256
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.basic_cnn.hdf5',
verbose=1, save_best_only=True)
start = datetime.now()
model.fit(x_train, y_train, batch_size=num_batch_size, epochs=num_epochs,
validation_data=(x_test, duration = datetime.now() - start
print("Training completed in time: ", duration)
```

```
Train on 6985 samples, validate on 1747 samples
Epoch 00070: val_loss did not improve from 0.27239
Epoch 71/72
6985/6985 [==============================] - 14289s 2s/step - loss: 0.1203 - acc:
0.9581 - val_loss: Epoch 00071: val_loss did not improve from 0.27239
Epoch 72/72
6985/6985 [==============================] - 92s 13ms/step - loss: 0.1147 - acc:
0.9596 - val_loss: Epoch 00072: val_loss did not improve from 0.27239
Training completed in time: 8:57:38.203486
```

3.3.4 Test the model

Here we will review the accuracy of the model on both the training and test data sets.

```
# Evaluating the model on the training and testing set
score = model.evaluate(x_train, y_train, verbose=0)
print("Training Accuracy: ", score[1])
score = model.evaluate(x_test, y_test, verbose=0)
print("Testing Accuracy: ", score[1])
```

```
Training Accuracy: 0.9819613457408733
Testing Accuracy: 0.9192902116210514
```

The Training and Testing accuracy scores are both high and an increase on our initial model. Training accuracy has increased by ~6% and Testing accuracy has increased by ~4%. There is a marginal increase in the difference between the Training and Test scores (~6% compared to ~5% previously) though the difference remains low so the model has not suffered from overfitting.

3.3.5 Predictions

Here we will modify our previous method for testing the models predictions on a specified audio .wav file.

```
def print_prediction(file_name):
prediction_feature = extract_features(file_name)
prediction_feature = prediction_feature.reshape(1, num_rows, num_columns, num_channels)
predicted_vector = model.predict_classes(prediction_feature)
predicted_class = le.inverse_transform(predicted_vector)
print("The predicted class is:", predicted_class[0], '\n')
predicted_proba_vector = model.predict_proba(prediction_feature)
predicted_proba = predicted_proba_vector[0]
for i in range(len(predicted_proba)):
category = le.inverse_transform(np.array([i]))
print(category[0], "\t\t : ", format(predicted_proba[i], '.32f') )
```

3.3.6 Validation

Test with sample data As before we will verify the predictions using a subsection of the sample audio files we explored in the first notebook. We expect the bulk of these to be classified correctly.

```
# Class: Air Conditioner
filename = '../UrbanSound Dataset sample/audio/100852-0-0-0.wav'
print_prediction(filename)
```

```
The predicted class is: air_conditioner
```

```
# Class: Drilling
filename = '../UrbanSound Dataset sample/audio/103199-4-0-0.wav'
```

```
print_prediction(filename)
The predicted class is: drilling
# Class: Street music
filename = '../UrbanSound Dataset sample/audio/101848-9-0-0.wav'
print_prediction(filename)
The predicted class is: street_music
# Class: Car Horn
filename = '../UrbanSound Dataset sample/audio/100648-1-0-0.wav'
print_prediction(filename)
The predicted class is: drilling
```

Observations We can see that the model performs well.
Interestingly, car horn was again incorrectly classifed but this time as drilling - though the per class confidence shows it was a close decision between car horn with 26% confidence and drilling at 34% confidence.

3.3.7 Other audio

Again we will further validate our model using a sample of various copyright free sounds that we not part of either our test or training data.

```
filename = '../Evaluation audio/dog_bark_1.wav'
print_prediction(filename)
The predicted class is: dog_bark
filename = '../Evaluation audio/drilling_1.wav'
print_prediction(filename)
The predicted class is: jackhammer
filename = '../Evaluation audio/gun_shot_1.wav'
print_prediction(filename)
The predicted class is: gun_shot
```

4 Results

4.1 Model Evaluation and Validation

During the model development phase the validation data was used to evaluate the model. The final model architecture and hyperparameters were chosen because they performed the best among the tried combinations. This architecture is described in detail in section 3.
As we can see from the validation work in the previous section, to verify the robustness of the final model, a test was conducted using copyright free sounds from sourced from the internet.
The following observations are based on the results of the test:
• The classifier performs well with new data.
• Misclassification does occur but seems to be between classes that are relatively similar such as Drilling and Jackhammer.

4.2 Justification

The final model achieved a classification accuracy of 92% on the testing data which exceeded my expectations given the benchmark was 68%.

Model	Classification Accuracy
CNN	92%
MLP	88%
Benchmark SVM_rbf	68%

The final solution performs well when presented with a .wav file with a duration of a few seconds and returns a reliable classification.
However, we do not know how the model would perform on Real-time audio. We do not know whether it would be able to perform the classification in a timely manner so audio frames are not skipped or the classification would be heavily affected by latency.
Also, we do not know how the classifier would perform in a real world setting. Our study makes no attempt to determine the effect of factors such as noise, echos, volume and salience level of the sample.

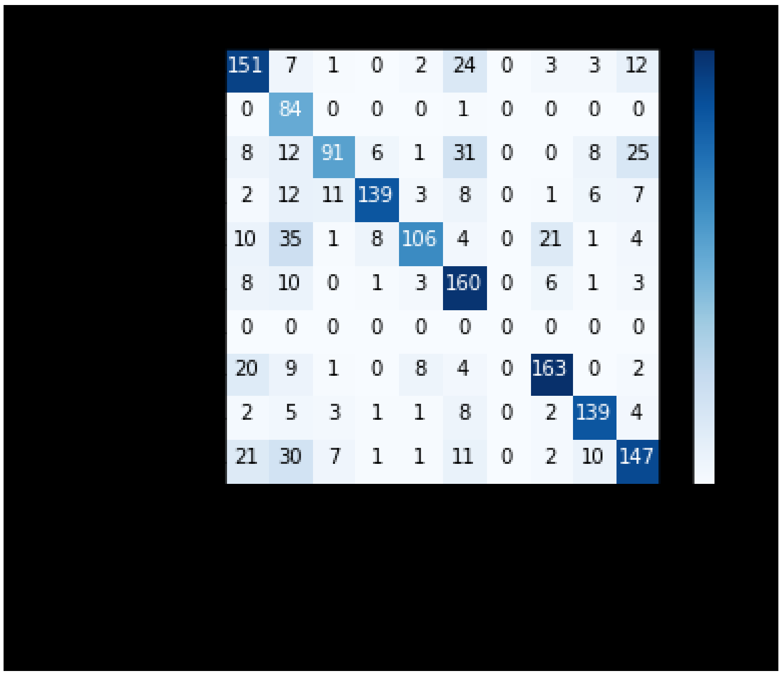
5 Conclusion

5.1 Free-Form Visualisation

It was previously noted in our data exploration, that it is difficult to visualise the difference between some of the classes. In particular, the following sub-groups are similar in shape:

• Repetitive sounds for air conditioner, drilling, engine idling and jackhammer.
• Sharp peaks for dog barking and gun shot.
• Similar pattern for children playing and street music.

Using a confusion matrix we will examine if the final model also struggled to differentiate between these classes.



151	7	1	0	2	24	0	3	3	12
0	84	0	0	0	1	0	0	0	0
8	12	91	6	1	31	0	0	8	25
2	12	11	139	3	8	0	1	6	7
10	35	1	8	106	4	0	21	1	4
8	10	0	1	3	160	0	6	1	3
0	0	0	0	0	0	0	0	0	0
20	9	1	0	8	4	0	163	0	2
2	5	3	1	1	8	0	2	139	4
21	30	7	1	1	11	0	2	10	147

The Confusion Matrix tells a different story. Here we can see that our model struggles the most
with the following sub-groups:

• air conditioner, jackhammer and street music.
• car horn, drilling, and street music.
• air conditioner, children playing and engine idling.
• jackhammer and drilling.
• air conditioner, car horn, children playing and street music.

This shows us that the problem is more nuanced than our initial assessment and gives some insights
into the features that the CNN is extracting to make it's classifications. For example, street music is one of the commonly classified classes and could be to a wide variety of different samples
within the class.

5.2 Reflection

The process used for this project can be summarised with the following steps:

1. The initial problem was defined and relevant public dataset was located.
2. The data was explored and analysed.
3. Data was preprocessed and features were extracted.
4. An initial model was trained and evaluated.

5. A further model was trained and refined.
6. The final model was evaluated.

From the initial exploration of the data in step 2, I envisaged that the preprocessing work in step
3 would be incredibly time consuming. However, this was actually relatively easy thanks to the

Python tool Librosa. I also thought that the feature extraction would be a lot trickier but again Librosa shortened the effort required immensely.
MFCC's we extracted in step 3 perform much better than I had expected. However, we had to revisit the extraction process when we transitioned to using a CNN as our model. I did consider
revisiting our MLP model to see how it performed with the updated feature extraction technique,
but unfortunately there was not enough time for this.
Overall, the model performed better than planned. One observation we made during step 2 is that
the dataset is slightly unbalanced with 2 out of the 10 classes having roughly a 40% sample size of
the other 8. However, it is unclear whether this is significant enough to have caused any issues.

5.3 Improvement

If we were to continue with this project there are a number of additional areas that could be explored:

- As previously mentioned, test the models performance with Real-time audio.
- Train the model for real world data. This would likely involve augmenting the training data in various ways such as:
  - Adding a variety of different background sounds.
  - Adjusting the volume levels of the target sound or adding echos.
  - Changing the starting position of the recording sample, e.g. the shape of a dog bark.
- Experiment to see if per-class accuracy is affected by using training data of different durations.
- Experiment with other techniques for feature extraction such as different forms of Spectrograms.

References

[1] Justin Salamon, Christopher Jacoby and Juan Pablo Bello. Urban Sound Datasets. https://urbansounddataset.weebly.com/urbansound8k.html
[2] Mel-frequency cepstrum Wikipedia page https://en.wikipedia.org/wiki/Mel-frequency_cepstrum
[3] J. Salamon, C. Jacoby, and J. P. Bello. A dataset and taxonomy for urban sound research. http://www.justinsalamon.com/uploads/4/3/9/4/4394963/salamon_urbansound_acmmm14.pdf
[4] Manik Soni AI which classifies Sounds:Code:Python. https://hackernoon.com/ai-which-classifies-sounds-code-python-6a07a2043810

[5] Manash Kumar Mandal Building a Dead Simple Speech Recognition Engine using ConvNet in Keras. https://blog.manash.me/building-a-dead-simple-word-recognition-engine-using-convnet-in-keras-25e72c19c12b
[6] Eijaz Allibhai Building a Convolutional Neural Network (CNN) in Keras. https://towardsdatascience.com/building-a-convolutional-neural-network-cnn-in-keras-329fbbadc5f5
[7] Daphne Cornelisse An intuitive guide to Convolutional Neural Networks. https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050
[8] Urban Sound Classification - Part 2: sample rate conversion, Librosa. https://towardsdatascience.com/urban-sound-classification-part-2-sample-rate-conversion-librosa-ba7bc88f209a
[9] wavsource.com THE Source for Free Sound Files and Reviews. http://www.wavsource.com/
[10] soundbible.com. http://soundbible.com//